S.A.N.I.C.E

(Sistema Automatizado de Normalização, Inteligência Computacional e Estatística)

S.A.N.I.C.E

(Sistema Automatizado de Normalização, Inteligência Computacional e Estatística)

Data Science Ágil para Quem Tem Pressa

Do CSV sujo ao Deploy de IA em poucas linhas

1ª Edição 2026

wSanice

Copyright © 2026 por wSanice Todos os direitos reservados.

Nenhuma parte deste livro pode ser reproduzida ou transmitida de qualquer forma ou por qualquer meio, eletrônico ou mecânico, incluindo fotocópia, gravação ou por qualquer sistema de armazenamento e recuperação de informações, sem a permissão por escrito do autor.

Isenção de Responsabilidade: As informações contidas neste livro são baseadas na experiência do autor e na documentação disponível até a data de publicação. O autor e o editor não assumem qualquer responsabilidade por erros ou omissões, ou por danos resultantes do uso das informações contidas aqui. O código fornecido é para fins educacionais e deve ser testado em ambiente seguro antes do uso em produção.

Dados Internacionais de Catalogação na Publicação (CIP)
Sanice, w. Sanice: Data Science Ágil para Quem Tem Pressa / wSanice. — 1. ed. — São Paulo : Edição do Autor, 2026.

1. Python (Linguagem de programação). 2. Ciência de Dados. 3. Inteligência Artificial. 4. Aprendizado de Máquina (Machine Learning). 5. Engenharia de Dados. 6. AutoML. 7. Bibliotecas Open Source. I. Título.

Índices para catálogo sistemático:

1. Computação : Ciência de Dados
2. Inteligência Artificial : Processamento de Dados

Versão da Biblioteca abordada: v1.0.10+ **Repositório do Projeto:** https://github.com/wSanice **Contato:** https://www.linkedin.com/in/wantruil-sanice/

A todos os desenvolvedores que já perderam uma tarde inteira tentando converter uma coluna de datas no Pandas. Este livro é para devolver o seu tempo.

E para minha esposa. Obrigado por caminhar ao meu lado. Meu coração é seu, hoje e em todas as linhas da nossa história.

"Fé cega é a ferramenta de monstros e tolos. Análise os dados."

— wSanice

S.A.N.I.C.E.....0
Nota do Autor: Por que mais uma biblioteca?.....13
PARTE 1.....14
OS FUNDAMENTOS.....14
Capítulo 1.....16
A Filosofia Fluente.....16
O Jeito Antigo vs. O Jeito Sanice.....16
Baterias Inclusas.....18
Preparando o Terreno.....19
A Era do Encadeamento (The Chaining Era).....20
A Anatomia do Caos.....21
O Jeito Pandas (10 Linhas de Tensão).....24
O Jeito Sanice (1 Linha Fluida).....25
A Diferença Invisível.....25
Capítulo 2.....26
Hello World dos Dados.....26
Instalando o Sanice (Com e sem baterias inclusas).....27
A Instalação "Cientista de Dados" (Recomendada).....27
Validando a Instalação.....28
A Regra de Ouro: Logs e Idiomas.....28
Controlando a Verbosidade.....28
O que acabou de acontecer?.....31
PARTE 2.....32
A ARTE DA FAXINA (ETL).....32
Capítulo 3.....34
Domando Dados Selvagens.....34
Como o Sanice resolve.....35
A mágica do .transformar() para Dinheiro.....36
Outras Regras de Transformação.....37
A Guilhotina ("apagar").....38
O Remendo ("preencher").....38
Capítulo 4.....41
Engenharia de Recursos (Feature Engineering).....41
Criando colunas com lógica matemática.....42

Lógica Condicional (Simples)..... 42
Filtros Inteligentes: A Sintaxe SQL no Python.....44
Otimização Automática (smart_run=True).....44
Prática: Para testar no vendas_limpas.csv (cap04_engenharia.py).....46
PARTE 3.....48
INTELIGÊNCIA ARTIFICIAL AUTOMATIZADA.....48
Capítulo 4.5..... 50
Visualização Expressa: O Fim do Matplotlib..... 50
O Método .plotar()..... 51
Prática: Olhando antes de saltar (cap4-5_visualizacao.py)... 53
Capítulo 5..... 56
O Torneio de Modelos (AutoML 2.0).....56
Os Competidores da Arena..... 57
Preparação Automática (O Trabalho Sujo Invisível).....58
O que esperar no Terminal?..... 61
Capítulo 6..... 62
Previsão na Prática..... 62
Carregando Cérebros Digitais (.carregar_ia)..... 62
Fazendo a Previsão (.prever)..... 63
Métricas de Sucesso: Posso confiar?..... 64
Vamos Brincar?..... 65
O que você deve fazer agora:..... 66
PARTE 4.....68
DO NOTEBOOK PARA O MUNDO.....68
Capítulo 7..... 70
O Poliglota de Dados (SQL & NoSQL).....70
Lendo direto da Fonte (Sanice.de_sql)..... 70
Exportando para o Futuro (.exportar_mongo).....72
Prática: O Pipeline de Integração.....72
Capítulo 8..... 75
Sua IA no Ar: Criando uma API.....75
O Servidor de Uma Linha (.servir_api).....76
O Código Final.....76

Testando sua API no Navegador....77
Fazendo uma Previsão Real....77
PARTE 5....80
Capítulo 9....82
Mordor Inc. vs Airbnb....82
Parte 1: Análise Exploratória....82
Parte 2: A Previsão....85
Conclusão: A Decisão do Lorde das Trevas....88
Capítulo 10....90
O Oráculo de Bolso (Deploy)....90
Solução:....91
A Vantagem do Cérebro Persistente:....94
Conclusão....95
Do Caos à Inteligência....95
A Vantagem da Automação....95
O Próximo Passo é Seu....95
Como você pode fazer parte?....96
Onde Encontrar o Projeto....97

Nota do Autor: Por que mais uma biblioteca?

Data Science deveria ser divertido.

Deveria ser sobre descobrir padrões ocultos em vendas, prever o futuro de uma ação ou entender por que um cliente abandonou seu carrinho de compras ou quem sabe prever a loteria. Mas, na maioria das vezes, Data Science é sobre lutar contra a sintaxe.

Se você já usou Python para dados, conhece a dor. Você quer apenas ler um Excel e limpar os nomes das colunas. De repente, você se vê escrevendo cinco linhas de código `pandas`, importando `numpy`, lidando com `axis=1`, `inplace=True` e erros de SettingWithCopyWarning.

Quando chegamos na parte da Inteligência Artificial, a situação piora. O Scikit-Learn é maravilhoso, mas exige que você trate nulos, converta texto em números (encoding), separe treino e teste, escale os dados e instancie o modelo... tudo isso antes mesmo de fazer a primeira previsão.

Eu criei o ***Sanice*** *porque eu tinha pressa.*

Eu queria uma ferramenta que respeitasse o meu tempo. Eu queria poder dizer ao computador: "Leia este arquivo, limpe os nomes das colunas, converta 'preço' para dinheiro e treine uma IA para mim". E eu queria dizer isso em uma única sentença.

O Sanice não nasceu para substituir o Pandas ou o Scikit-Learn; ele nasceu para orquestrá-los. Ele é um wrapper fluente, opinativo e robusto, projetado para levar você do dado bruto ao modelo em produção na velocidade do pensamento.

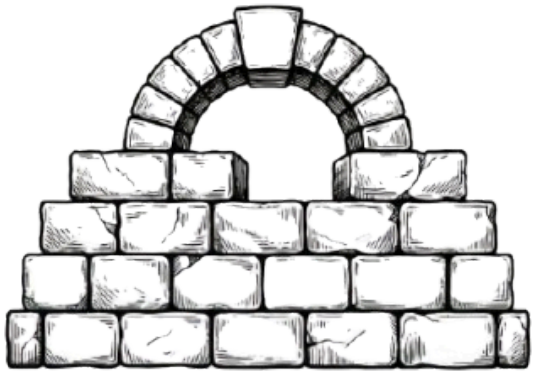
Este livro não é um manual acadêmico. É um guia de campo. Nas próximas páginas, não vamos discutir a matemática profunda por trás do Gradient Descent. Vamos discutir como colocar um modelo de previsão de preços no ar antes da hora do almoço.

Se você tem dados sujos e prazos apertados, você está no lugar certo.

Vamos codar.

wSanice *São Paulo, 2026*

PARTE 1



OS FUNDAMENTOS

Capítulo 1

A Filosofia Fluente

"A simplicidade é o último grau de sofisticação." — Leonardo da Vinci

Imagine que você está em um restaurante. Você não diz ao garçom: "Caminhe até a cozinha. Pegue um prato. Coloque arroz. Coloque feijão. Volte até a mesa."

Você apenas diz: "Eu quero o prato feito."

Na programação, especialmente em Data Science, costumamos ser excessivamente imperativos. Passamos mais tempo dizendo **como** o computador deve fazer algo do que **o que** queremos que ele faça.

O Sanice foi construído sobre a filosofia da **Interface Fluente** (ou Method Chaining). A ideia é que o código deve ser lido como uma frase em linguagem natural, onde o resultado de uma operação é passado imediatamente para a próxima.

O Jeito Antigo vs. O Jeito Sanice

Vamos ver um exemplo clássico. Você tem um arquivo `vendas.csv` com colunas bagunçadas (acentos, espaços) e uma coluna de preço que veio como texto (`"R$ 1.200,00"`). Você quer limpar isso e remover linhas vazias.

O Jeito Tradicional (Pandas Puro):

```
import pandas as pd
import numpy as np

# 1. Carregar
df = pd.read_csv("vendas.csv")

# 2. Limpar nomes de colunas (Manual e doloroso)
cols_novas = []
for c in df.columns:
        cols_novas.append(c.lower().replace(" ",
"_").replace("ç", "c").replace("ã", "a")) df.columns =
cols_novas

# 3. Converter dinheiro (Regex complexo)
df['valor'] = df['valor'].astype(str).str.replace('R$', '',
regex=False)
df['valor'] = df['valor'].str.replace('.', '',
regex=False).str.replace(',', '.')
df['valor'] = pd.to_numeric(df['valor'])

# 4. Remover nulos
df = df.dropna()
```

São mais de 10 linhas de código. Se você errar uma vírgula no `replace`, o script quebra. Agora, veja a mesma operação com a filosofia do Sanice.

O Jeito Sanice:

```
from sanice import Sanice

(Sanice("vendas.csv")
        .corrigir_colunas()
        .transformar("valor", "dinheiro")
        .remover_nulos()
        .ver()
)
```

Quatro linhas. Nenhuma variável temporária (`df = ...`). Nenhuma Regex complexa visível. Apenas verbos de ação: **Corrigir**, **Transformar**, **Remover**.

Isso é o que chamamos de **Abstração Produtiva**. O código complexo do Pandas ainda está lá, rodando nos bastidores, mas você não precisa gerenciá-lo. Você foca na regra de negócio.

Baterias Inclusas

Outro pilar da filosofia Sanice é que você não deve precisar caçar bibliotecas para tarefas básicas.

- Quer conectar no banco de dados? O Sanice já traz o driver.
- Quer criar uma API? O Sanice já traz o servidor.
- Quer plotar um gráfico? O Sanice já fala com o Matplotlib.

Nas versões mais recentes (v1.0.9+), adotamos a "Lei da Importação Única". Ao importar o `Sanice`, você ganha acesso a todo o ecossistema necessário:

```
# Tudo o que você precisa em uma linha
from sanice import Sanice, pd, plt, joblib
```

Isso elimina aquele cabeçalho gigante de `imports` que assusta iniciantes e polui scripts de profissionais.

Preparando o Terreno

Antes de começarmos a limpar terabytes de dados, precisamos preparar nosso ambiente. No próximo capítulo, vamos instalar a biblioteca e rodar nosso primeiro "Hello World" com dados reais.

Prepare seu terminal.

O fim do código espaguete e a era do encadeamento

Em 1968, Edsger Dijkstra publicou uma carta famosa intitulada "Go To Statement Considered Harmful", criticando o uso desenfreado de comandos de "pulo" no código, que criavam uma lógica emaranhada impossível de seguir o temido **Código Espaguete**.

Cinquenta anos depois, os Cientistas de Dados recriaram o código espaguete dentro dos Jupyter Notebooks.

Se você olhar para um notebook típico de análise de dados, verá algo assim:

1. Carrega-se um `df`.
2. Cria-se um `df_limpo`.
3. Cria-se um `df_sem_nulos`.
4. Lá na célula 15, altera-se o `df` original.

5. Na célula 20, tenta-se usar o `df_limpo`, mas ele já não reflete a alteração da célula 15.

O resultado é um estado global confuso, onde você precisa rodar as células na ordem exata (e lembrar qual variável guarda qual versão dos dados) para que o script funcione. Isso é frágil. Isso é perigoso em produção.

A Era do Encadeamento (The Chaining Era)

A solução para isso não é nova, mas foi popularizada em Data Science pelo dplyr (no R) e agora pelo Sanice (no Python). Chama-se **Method Chaining** (Encadeamento de Métodos).

A ideia é tratar o processamento de dados como uma tubulação industrial (pipeline). O dado entra bruto por um lado, passa por uma série de transformações sequenciais e sai refinado do outro lado.

No Sanice, quase todo método retorna o próprio objeto (`return self`), carregando o estado modificado. Isso permite que você escreva código que se lê de cima para baixo, como uma lista de instruções clara:

```
# O estilo Sanice: Uma única sentença fluida
(Sanice("dados.csv")
        .limpar()
        .filtrar()
        .treinar()
)
```

Não há variáveis intermediárias perdidas. Não há "estado oculto". O código conta uma história linear. Se houver um erro no passo 2, o passo 3 nunca acontece. Isso traz previsibilidade e paz de espírito para o desenvolvedor.

O problema da verbosidade no Pandas tradicional

O Pandas é, sem dúvida, a biblioteca de manipulação de dados mais importante do mundo Python. Ele é o motor V8 que faz tudo funcionar. O Sanice não substitui o motor do Pandas; ele troca o volante e o painel.

O problema do Pandas é que sua API foi desenhada para ser explícita e flexível, o que muitas vezes resulta em **verbosidade excessiva** para tarefas triviais.

Vamos analisar a anatomia de uma operação simples: limpar uma coluna de preços (remover "R$" e converter para número).

A Anatomia do Caos

No Pandas puro ("Vanilla Pandas"), você precisa repetir o nome do DataFrame constantemente:

```
# O Jeito "Vanilla" (Pandas Puro)
# 1. Repetição do nome do objeto
df['preco'] = df['preco'].astype(str)

# 2. Sintaxe complexa para operações simples
df['preco'] = df['preco'].str.replace('R$', '', regex=False)

# 3. Gerenciamento de tipos manual
df['preco'] = pd.to_numeric(df['preco'])
```

Note que escrevemos `df['preco']` **quatro vezes** para realizar uma única intenção lógica. Isso aumenta a "carga cognitiva". Seus olhos precisam escanear o código para garantir que você não escreveu `df['prazo']` por engano em uma das linhas.

A Solução Fluente

O Sanice encapsula essa complexidade. Ele assume que, se você quer transformar uma coluna, você quer que o resultado persista no DataFrame.

```
# O Jeito Sanice

app.transformar("preco", "dinheiro")
```

Ao encapsular a lógica repetitiva (como `str.replace` e `to_numeric`) dentro de "Regras de Negócio" (como `"dinheiro"`, `"email"`, `"data"`), removemos o ruído sintático.

Você para de ler **como** o código funciona (regex, astype, replace) e passa a ler **o que** o código faz.

> **Nota Técnica:** Por baixo do capô, o Sanice está usando exatamente o Pandas e o NumPy que você conhece. A performance é a mesma. A diferença é que *você escreve em 5 segundos o que levaria 5 minutos de pesquisa no StackOverflow para lembrar a sintaxe correta do* ***`regex`***.

Entendendo o Method Chaining (Interface Fluente)

Se você abrir o código-fonte do Sanice, notará um padrão curioso repetido em quase todos os métodos. Do `.limpar_texto()` ao `.auto_ml()`, todos eles terminam com a mesma instrução mágica:

```
return self
```

Essa simples linha é o segredo por trás da **Interface Fluente**.

Na programação orientada a objetos tradicional, muitos métodos são "void" (não retornam nada) ou retornam um valor novo, desconectado do objeto original. Isso força você a parar o fluxo para salvar o resultado em uma variável.

No Sanice, adotamos o padrão de que **o objeto é o contexto**.

Imagine uma linha de montagem industrial. O chassi do carro (o `DataFrame`) entra na esteira. O primeiro robô solda as portas (`.unir()`). O segundo robô pinta a lataria (`.transformar()`). O terceiro coloca as rodas (`.tratar_outliers()`).

Em nenhum momento o carro é retirado da esteira, colocado no chão do depósito, renomeado para `carro_pintado`, colocado de volta na esteira, depois retirado de novo e renomeado para `carro_com_rodas`. Isso seria ineficiente e propenso a erros de logística.

No entanto, é exatamente isso que fazemos quando escrevemos:

df = pd.read_csv(...)

df_limpo = df.dropna()

df_final = df_limpo.merge(...)

O Fluxo da Leitura Humana

O Method Chaining alinha o código à forma como o cérebro humano processa listas de tarefas. Quando lemos um bloco encadeado, lemos:

1. **Sujeito:** Sanice ("Pegue este arquivo")
2. **Verbo 1:** Corrigir Colunas ("Padronize os nomes")
3. **Verbo 2:** Transformar ("Limpe esta moeda")
4. **Verbo 3:** Salvar ("Guarde o resultado")

A carga cognitiva diminui drasticamente. Você não precisa lembrar se a variável se chama `df` ou `df_clean`. Você não precisa rastrear o estado. Você confia no fluxo.

Comparativo: 10 linhas vs. 1 linha

Para encerrar este capítulo e provar que não estamos falando apenas de estética, vamos para um duelo prático.

O Cenário: Você recebeu um arquivo `vendas_brutas.csv`.

1. Os nomes das colunas têm acentos e espaços ("Data da Venda", "Preço Unitário").
2. A coluna de preço é texto ("R$ 1.500,00").
3. Existem linhas vazias que precisam ser removidas.
4. Você precisa salvar o resultado limpo.

O Jeito Pandas (10 Linhas de Tensão)

```
import pandas as pd
# 1. Carregar
df = pd.read_csv("vendas_brutas.csv")
# 2. Limpar Colunas (Manual)
df.columns = [c.strip( ).lower( ).replace("", "_")
              .replace("ç", "c").replace("ã", "a")
              for c in df.columns]
# 3. Limpar Moeda (Regex assustador)

df['preco_unitario'] = df['preco_unitario'].astype(str)
```

```
df['preco_unitario'] = df['preco_unitario'].str.replace('R$', ' ',
regex=False)
df['preco_unitario'] = df['preco_unitario'].str.replace('.', ' ',
regex=False)
df['preco_unitario'] = df['preco_unitario'].str.replace(',', '.')
df['preco_unitario'] = pd.to_numeric(df['preco_unitario'])
# 4. Salvar
df.dropna().to_csv("vendas_limpas.csv", index=False)
```

Se você esquecer o `regex=False` ou trocar a ordem do replace da vírgula pelo ponto, você corrompe os dados silenciosamente (transformando 1.500 em 1,5).

O Jeito Sanice (1 Linha Fluida)

```
from sanice import Sanice
(Sanice("vendas_brutas.csv")
   .corrigir_colunas()
   .transformar("preco_unitario", "dinheiro")
   .remover_nulos()
   .salvar("vendas_limpas.csv")
)
```

A Diferença Invisível

A diferença óbvia é a quantidade de código escrito. Mas a diferença real é a **abstração da complexidade**.

No exemplo do Sanice, você não precisou dizer ao computador como remover o "R$" ou como lidar com a vírgula decimal brasileira. Você declarou sua **intenção** (`"dinheiro"`), e a biblioteca lidou com a implementação técnica nos bastidores.

Isso libera seu cérebro para focar no que realmente importa: **O que esses dados dizem sobre o meu negócio?**

Agora que você entendeu a filosofia, vamos sujar as mãos. No próximo capítulo, faremos nosso "Hello World".

Capítulo 2

Hello World dos Dados

"A jornada de mil terabytes começa com um único `pip install`."

Em qualquer linguagem de programação, o rito de passagem é fazer o computador dizer "Olá". Em Data Science, nosso "Hello World" é um pouco mais exigente: queremos carregar um arquivo, ver o que tem dentro e não explodir a memória RAM no processo.

Neste capítulo, vamos configurar seu ambiente de trabalho.

Muitos livros gastam 50 páginas explicando como instalar o Anaconda, configurar ambientes virtuais (`venvs`) e lidar com variáveis de ambiente do Windows. Nós não faremos isso.

Eu assumo que você já tem o **Python 3.8+** instalado. Se não tem, resolva isso e volte aqui. O Sanice não julga se você usa VS Code, PyCharm ou o Bloco de Notas (embora eu julgue um pouco se for o Bloco de Notas).

O objetivo aqui é sair do zero e ter um pipeline de dados funcional em menos de 5 minutos.

Instalando o Sanice (Com e sem baterias inclusas)

A filosofia do Sanice é "Baterias Inclusas". Isso significa que você não deveria ter que instalar o driver do banco de dados separadamente, depois a biblioteca de gráficos, depois a de IA.

No entanto, entendemos que em ambientes de produção (como um container Docker minimalista), você pode querer apenas o essencial. Por isso, oferecemos dois sabores de instalação.

A Instalação "Cientista de Dados" (Recomendada)

Se você está na sua máquina local e quer todo o poder de fogo (conectar no Mongo, gerar APIs, plotar gráficos), use este comando no seu terminal:

```
pip install "sanice[api,db]"
```

Ao adicionar `[api,db]`, o pip trará automaticamente o **FastAPI** (para o comando `.servir_api()`) e drivers como `pymongo` e `psycopg2` (para SQL e NoSQL).

A Instalação "Minimalista"

Se você quer apenas limpar dados e salvar CSVs, ou se sua internet é discada, instale apenas o núcleo:

```
pip install sanice
```

Isso trará apenas o Pandas, NumPy e Scikit-Learn. O framework é inteligente o suficiente para não quebrar se você tentar usar um recurso ausente; ele gentilmente pedirá que você instale a dependência que falta.

Validando a Instalação

Não precisa abrir o Python para saber se funcionou. O Sanice vem com uma ferramenta de linha de comando (CLI) embutida. Digite no terminal:

```
sanice ajuda
```

Se você vir uma linha começando com `=== Sanice CLI Help (PT) ===` e abaixo uma lista de comandos em Português e mais 3 línguas, parabéns. Você está pronto.

A Regra de Ouro: Logs e Idiomas

Antes de escrevermos código, precisamos falar sobre **Controle**.

Bibliotecas de Data Science costumam ser mudas (você roda e não sabe se travou) ou tagarelas demais (imprimindo centenas de avisos de deprecation).

O Sanice adota uma postura de "Feedback Transparente". Por padrão, ele conta o que está fazendo: "Carreguei 1000 linhas", "Removi 50 nulos". Mas em produção, você quer silêncio.

Controlando a Verbosidade

Você controla o "volume" do Sanice com o método `.configurar_logs()`:

- `"info"` **(Padrão):** O Sanice narra o jogo. Útil para desenvolvimento e notebooks.
- `"silent"` **(Modo stealth):** O Sanice fica mudo. Essencial para quando seu script vai rodar num servidor (cronjob) ou virar uma API JSON pura.

```
# Modo Desenvolvimento (Padrão)
app.configurar_logs("info")
# Saída: [CARREGAR] Dados carregados... [LIMPEZA] Nomes padronizados...

# Modo Produção
app.configurar_logs("silent")
# Saída: (Vazio)
```

O Poliglota (Internacionalização)

O código é em inglês (Python), mas os dados e o time podem não ser. O Sanice permite que você programe pensando em até 4 idiomas

Ao instanciar a classe, o parâmetro `lang` define duas coisas críticas:

1. A língua dos logs no terminal.
2. A moeda padrão para limpeza (BRL, USD, CNY, INR).

Isso é sutil, mas poderoso. Evita que você limpe uma coluna de Dólares achando que são Reais, apenas porque esqueceu de configurar a localidade.

Seu primeiro script: Carregar, Ver e Resumir

(`ola_dados.py`.)

Vamos colocar a mão na massa. Crie um arquivo chamado **`ola_dados.py`**.

Vamos usar um recurso introduzido na versão v1.0.9: a **Importação Unificada**. Em vez de importar pandas, matplotlib e joblib separadamente, importamos tudo do Sanice. Isso mantém seu cabeçalho limpo.

```
from sanice import Sanice

# 1. Instanciamos apontando para um arquivo (pode ser CSV, Excel, JSON ou Parquet)
# O "smart_run=True" já tenta detectar datas e otimizar memória RAM.
app = Sanice("../data/raw/vendas_jan_2026.csv", lang="pt", smart_run=True)

# 2. O Pipeline de Reconhecimento
(app
   .ver(5)                # Mostra as primeiras 5 linhas
   .resumo_estatistico()  # Mostra média, min, max e tipos de dados
)
```

O que acabou de acontecer?

Ao rodar **`python ola_dados.py`**, o terminal ganhará vida:

```
[CARREGAR] Dados carregados: 15000 linhas x 12 colunas.
[SMART] Memória otimizada! 3 colunas convertidas para 'category'.
[SMART] Data detectada e convertida na coluna: 'data_venda'

[VISUALIZAR] Top 5:
   id_venda data_venda  valor   produto
0      101 2025-01-01  150.0  Teclado
...

[ESTATÍSTICAS] Resumo Estatístico:
         count   mean    std ...
valor   15000.0  145.2   12.5 ...
```

Sem que você pedisse explicitamente, o `smart_run=True`:

1. Leu o arquivo.
2. Percebeu que a coluna `data_venda` parecia uma data e a converteu (adeus `pd.to_datetime`).
3. Percebeu que a coluna `produto` tinha poucos valores únicos repetidos e a converteu para Categoria, economizando RAM.

Você economizou cerca de 10 linhas de código de "boilerplate" (código repetitivo) apenas na carga dos dados.

Agora que vimos os dados, vamos começar a limpá-los. Prepare-se, porque no próximo capítulo vamos atacar a parte mais suja do trabalho: **A Faxina (ETL).**

PARTE 2

A ARTE DA FAXINA (ETL)

Capítulo 3

Domando Dados Selvagens

"Dados limpos superam algoritmos inteligentes. Dados sujos enganam até gênios."

Você carregou seu arquivo ***`vendas_jan_2026.csv`***. O terminal mostrou que ele existe e tem dados. Vitória. Mas agora a realidade bateu à porta.

Ao olhar com atenção, você percebe os problemas:

1. As colunas têm nomes inconsistentes como `frete_formato_sujo` ou `cidade_cliente` (às vezes com acentos ou espaços em outros datasets).
2. A coluna de frete, que deveria ser um número para somarmos, está cheia de textos como `"R$ 25,00"` misturados com células vazias.
3. Nomes de cidades estão uma bagunça: "sao paulo", "Sao Paulo " (com espaço no fim) e "RIO DE JANEIRO".

Se você tentar calcular a média de frete agora, o Python vai gritar um erro. Para o computador, `"R$ 25,00"` é texto, igual a `"Banana"`.

Neste capítulo, vamos transformar esse caos em ordem usando a tríade de limpeza do Sanice: **Corrigir**, **Transformar** e **Remover**.

O comando `.corrigir_colunas()`: Higiene Básica

O primeiro inimigo da produtividade é o nome da coluna.

Imagine ter que escrever `df['Preço Unitário (R$)']` toda vez que quiser acessar o preço. Além de cansativo, é propenso a erros. Um espaço a mais e seu código quebra.

O padrão ouro em Data Science é o **snake_case**: tudo minúsculo, sem acentos, com palavras separadas por underline.

Como o Sanice resolve

Em vez de renomear coluna por coluna manualmente, usamos o método `.corrigir_colunas()`. Ele varre seu cabeçalho e aplica uma higienização pesada:

1. **Remove acentos:** *`Vitória`* vira *`Vitoria`*.
2. **Remove caracteres especiais:** *`Preço($)`* vira *`Preco`*.
3. **Padroniza espaços:** *`Data Venda`* vira *`data_venda`*.
4. **Caixa Baixa:** Tudo vira minúsculo.

```
# ANTES:
# Colunas hipotéticas: ['Data Venda', 'Categoria', 'Frete (Estimado)']

app.corrigir_colunas()

# DEPOIS:
# Colunas: ['data_venda', 'categoria', 'frete_estimado']
```

No nosso dataset ***`vendas_jan_2026.csv`***, as colunas já vieram quase boas, mas rodar esse comando é uma "apólice de seguro". Garante que, se a fonte dos dados mudar amanhã e vier com acentos, seu código não quebra.

A mágica do `.transformar()` para Dinheiro

Agora vamos ao vilão clássico dos dados brasileiros: a formatação de moeda.

No nosso dataset, temos a coluna `frete_formato_sujo`. Alguns valores são `NaN` (nulos) e outros são textos como `"R$ 45,00"`.

Limpar isso no Pandas puro ("Vanilla") exigiria importar Regex (Expressões Regulares), substituir "R$", depois substituir o ponto de milhar por nada, e a vírgula decimal por ponto. É fácil errar.

O Sanice encapsula essa lógica na regra `"dinheiro"` (ou `"money"`).

```
# O Jeito Sanice
# A regra 'dinheiro' detecta R$, $, vírgulas e pontos automaticamente.
app.transformar("frete_formato_sujo", "dinheiro")
```

Ao rodar isso, o Sanice converte `"R$ 1.500,99"` para o float `1500.99`. Se ele encontrar um valor que não pode ser convertido (como um texto "Grátis"), ele o transforma em `NaN` para tratarmos depois.

Outras Regras de Transformação

O método `.transformar(colunas, regra)` é um canivete suíço. Além de dinheiro, ele aceita:

- `"email"`: Converte para minúsculo, remove espaços e valida se tem "@" e ".".
- `"numeros"`: Remove tudo o que não for dígito (ótimo para limpar CPF ou telefones como "(11) 9999-8888" -> "1199998888").
- `"upper"` / `"lower"`: Converte textos para MAIÚSCULO ou minúsculo.

Limpeza de Texto com *.limpar_texto()*

Nomes de cidades e produtos frequentemente vêm inconsistentes. Isso é mortal para agrupamentos. Se você agrupar vendas por cidade, o Python achará que "sao paulo" e "SAO PAULO" são lugares diferentes.

O comando `.limpar_texto()` padroniza strings para **Title Case** (Primeira Letra Maiúscula) e remove espaços em branco nas pontas (`strip`).

```
# Padronizando cidades e produtos no nosso dataset
app.limpar_texto(["cidade_cliente", "produto"])

# Resultado:
# "sao paulo "   -> "Sao Paulo"
# "RIO DE JANEIRO" -> "Rio De Janeiro"
```

Agora, seus gráficos de "Vendas por Cidade" serão precisos e unificados.

Estratégias para Nulos: `.remover_nulos()`

Depois de limpar moedas e textos, é normal sobrar "buraqueira" nos dados (valores `NaN`).

- Aquele frete que não conseguimos ler virou `NaN`.
- Algumas categorias vieram vazias da origem.

O Sanice oferece duas vias no método `.remover_nulos()`:

A Guilhotina ("apagar")

Se a linha tem dados faltando, ela não é confiável. Corta-se a linha inteira. É o padrão.

```
# Remove qualquer linha que tenha pelo menos um campo vazio
app.remover_nulos("apagar")
```

O Remendo ("preencher")

Às vezes, perder dados é um luxo que não temos. Se o frete está vazio, talvez possamos assumir que é zero (Frete Grátis).

```
# Preenche os buracos com 0.0
app.remover_nulos("preencher", 0.0)
```

Dica de Ouro: *A ordem importa! Geralmente, limpamos e ransformamos primeiro (o que pode gerar mais nulos ao falhar na conversão de textos inválidos) e removemos os nulos por último.*

Prática: O Script da Faxina Completa

(`cap03_faxina.py`)

Vamos juntar tudo em um script novo. Crie o arquivo ***`cap03_faxina.py`*** na pasta ***`chapters`***.

Este script vai pegar o arquivo sujo, aplicar tudo que aprendemos e salvar uma versão limpa na pasta ***`data/processed`*** (crie essa pasta se não existir).

```
import os
from sanice import Sanice

# 1. Definindo caminhos (Boas práticas)
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
CAMINHO_ENTRADA = os.path.join(BASE_DIR, "data", "raw", "vendas_jan_2026.csv")
CAMINHO_SAIDA = os.path.join(BASE_DIR, "data", "processed", "vendas_limpas.csv")

# Certifique-se que a pasta de saída existe
os.makedirs(os.path.dirname(CAMINHO_SAIDA), exist_ok=True)

# 2. O Pipeline de Limpeza
print("--- Iniciando a Faxina ---")

(Sanice(CAMINHO_ENTRADA)
   .corrigir_colunas() # Padroniza nomes (snake_case)
   .transformar("frete_formato_sujo", "dinheiro") # Limpa "R$ 25,00" -> 25.0
   .limpar_texto(["cidade_cliente", "produto"])  # "sao paulo " -> "Sao Paulo"
   .remover_nulos("apagar") # Remove linhas com falhas
   .ver(5, titulo="Dados Limpos")  # Confere o resultado
   .salvar(CAMINHO_SAIDA)  # Salva o arquivo pronto
)
```

```
print(f"\nSucesso! Arquivo limpo salvo em: {CAMINHO_SAIDA}")
```

Ao rodar isso, você verá no terminal os logs do Sanice trabalhando para você: `[TRANSFORMAR]`, `[LIMPEZA]`, `[REMOVER]`.

Agora que temos dados limpos, podemos começar a criar inteligência. No próximo capítulo, falaremos de **Feature Engineering**: como criar novas colunas que explicam o comportamento do seu cliente.

Capítulo 4

Engenharia de Recursos (Feature Engineering)

"O algoritmo é o motor, mas as features são o combustível."

Ter uma coluna `data_venda` limpa (ex: "2025-01-15") é bom. Mas, para um modelo de Inteligência Artificial prever vendas, ele não entende "15 de Janeiro". Ele entende conceitos matemáticos como:

- É fim de semana? (Sim/Não)
- Quantos dias faltam para o fim do mês?
- Qual é o valor total da compra (Preço x Quantidade + Frete)?

Feature Engineering é a arte de criar novas colunas que contam histórias melhores para a IA.

No Pandas tradicional, isso envolveria loops complexos ou funções lambda difíceis de ler. No Sanice, usamos verbos diretos: `.criar_coluna()`, `.converter_data()` e lógica simples.

Criando colunas com lógica matemática

Vamos começar com o básico: quanto dinheiro entrou nessa venda? Temos `valor` (unitário), `quantidade` e o `frete_formato_sujo` (que agora é um número limpo).

Podemos criar a coluna `total_venda` usando uma string de expressão matemática, muito parecida com o que você faria no Excel ou SQL.

```
# Criando o Total da Venda
# Fórmula: (Valor * Quantidade) + Frete
app.criar_coluna("total_venda", "valor * quantidade + frete_formato_sujo")
```

O Sanice usa o método `eval` do Pandas por baixo dos panos, o que torna essa operação extremamente rápida e legível.

Lógica Condicional (Simples)

E se quisermos classificar a venda? "Venda Grande" vs "Venda Pequena"? Podemos usar funções lambda do Python dentro do `.criar_coluna()`.

```
# Criando uma classificação de ticket
# Se total > 5000 é 'Gold', senão é 'Standard'

def classificar_ticket(linha):
    return "Gold" if linha["total_venda"] > 5000 else "Standard"

app.criar_coluna("tipo_cliente", classificar_ticket)
```

Agora seu dataset tem uma nova dimensão de análise categórica.

O Poder das Datas: `.converter_data()`

Datas são minas de ouro de informação, mas costumam vir como texto (`object`). O Sanice resolve isso com `.converter_data()`.

```
# 1. Converter texto para objeto de Data real
app.converter_data("data_venda")
```

Uma vez convertida, você pode extrair inteligência. Vamos criar colunas para "Dia da Semana" e "Semana do Ano". Como o Sanice expõe o DataFrame interno via `.df` (ou wrapper transparente), podemos usar o acessor `.dt` do Pandas.

Mas, para manter a filosofia fluente, recomendo criar colunas calculadas:

```
# Extraindo inteligência temporal
app.criar_coluna("dia_semana", lambda x:
x["data_venda"].day_name())
app.criar_coluna("eh_fim_de_semana", lambda x: 1 if
x["data_venda"].dayofweek >= 5 else 0)
```

Agora, sua IA poderá aprender padrões como: "As vendas de periféricos gamer aumentam aos sábados?"

Filtros Inteligentes: A Sintaxe SQL no Python

Muitas vezes, você não quer analisar a empresa inteira. Você quer ver apenas "Vendas do Rio de Janeiro acima de R$ 1.000".

No Pandas tradicional, a sintaxe de filtro é verbosa e repetitiva: `df = df[(df['estado'] == 'RJ') & (df['valor'] > 1000)]`

No Sanice, usamos o método `.filtrar()`*, que aceita uma string elegante estilo SQL:*

```
# Filtrando apenas RJ com vendas altas
app.filtrar("estado == 'RJ' and valor > 1000")
```

Isso torna a exploração de dados interativa e rápida. Você pode encadear filtros para "esculpir" seu dataset:

```
(app
   .filtrar("categoria == 'Perifericos'")
   .filtrar("quantidade >= 2")
   .ver(5, titulo="Vendas de Hardware em Volume")
)
```

Otimização Automática (`smart_run=True`)

Já mencionamos isso no Capítulo 2, mas vale reforçar. Quando você lida com Big Data (milhões de linhas), textos repetidos consomem muita memória RAM.

Se você tem 1 milhão de vendas e a coluna `estado` só tem 27 valores possíveis (SP, RJ, MG...), guardar isso como texto ("String") é um desperdício.

O parâmetro `smart_run=True` na inicialização do Sanice ativa o **Detector de Categorias**.

```
# Otimização silenciosa
app = Sanice("dados_gigantes.csv", smart_run=True)
```

Ele escaneia suas colunas. Se uma coluna tem poucos valores únicos (baixa cardinalidade) em relação ao total de linhas, ele converte para o tipo `category` do Pandas/C. Isso pode reduzir o uso de memória em até **80%** sem você mudar uma linha de código.

Tratando Outliers (Anomalias)

Outliers são dados que fogem do padrão e distorcem a realidade. Uma venda de R$ 1.000.000,00 num dataset onde a média é R$ 100,00 vai estragar sua IA.

O Sanice traz o método `.tratar_outliers()`, que usa a estatística do **Intervalo Interquartil (IQR)**. Basicamente, ele corta tudo o que está muito longe da média (os 1% extremos).

```
# Remove vendas com valores absurdamente altos ou baixos
app.tratar_outliers("valor")
```

Prática: Para testar no `vendas_limpas.csv`

`(cap04_engenharia.py)`

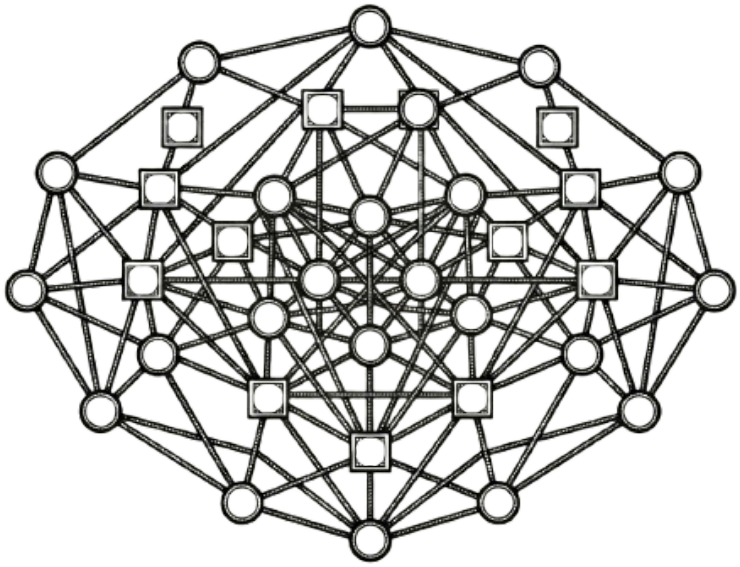
Crie o arquivo `cap04_engenharia.py` na pasta `chapters` para testarmos essas transformações nos dados que limpamos no capítulo anterior.

```
import os
from sanice import Sanice

# 1. Definindo Caminhos
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
ENTRADA = os.path.join(BASE_DIR, "data", "processed", "vendas_limpas.csv")
SAIDA = os.path.join(BASE_DIR, "data", "processed", "vendas_com_features.csv")

print("--- Iniciando Feature Engineering ---")

# 2. Carregar dados limpos
app = Sanice(ENTRADA)

(app
    # Converte string para Data Real (na memória)
   .converter_data("data_venda")

   # Criamos o Subtotal pré-calculado
   .criar_coluna("subtotal", "valor * quantidade")

   # Cria "total_venda" (Matemática: Qtd + Frete)
   .criar_coluna("total_venda", "subtotal + frete_formato_sujo")


   # Cria "comissao" (Regra de Negócio: 5% do total)
   .criar_coluna("comissao", "total_venda * 0.05")
```

```
    # Cria "eh_fim_de_semana" (Lógica Lambda: Sábado(5) ou Domingo(6) = 1)
    .criar_coluna("eh_fim_de_semana", lambda row: 1 if row["data_venda"].dayofweek >= 5 else 0)

    # Opcional: Ordenar para ver os maiores valores primeiro
    .ordenar("total_venda", ascendente=False)

    # Visualizar para garantir que funcionou
    .ver(5, titulo="Novas Features Criadas")

    # 3. SALVAR (CRUCIAL PARA O CAPÍTULO 5)
    # Grava o arquivo enriquecido no disco
    .salvar(SAIDA)
)

print(f"\nSucesso! Arquivo pronto para IA salvo em: {SAIDA}")
```

Rode esse script e veja como seu dataset ganha novas colunas ricas em informação (`total_venda, comissao, eh_fim_de_semana`). São essas colunas que alimentarão nossa IA no próximo capítulo.

PARTE 3



INTELIGÊNCIA ARTIFICIAL AUTOMATIZADA

Capítulo 4.5

Visualização Expressa: O Fim do Matplotlib

"Não me diga os números. Mostre-me o padrão."

Você limpou os dados. Você criou colunas inteligentes. Agora, sua intuição diz que está pronto para treinar a IA. Espere.

Um erro clássico de novatos é jogar dados "cegos" dentro de um algoritmo. Antes de modelar, você precisa ver.

Se você já tentou criar gráficos em Python, sabe que bibliotecas como `matplotlib` ou `seaborn` são poderosas, mas verbosas. Você escreve 15 linhas de código para definir o tamanho da figura, rotacionar o eixo X, colocar título e legenda. Quando o gráfico finalmente aparece, você já esqueceu o que queria analisar.

O Sanice resolve isso encapsulando as melhores bibliotecas de visualização (Plotly e Seaborn) em um único verbo: `.plotar()`.

O Método `.plotar()`

A filosofia aqui é a **Exploração Rápida**. Não queremos gráficos de arte para publicar na revista Nature; queremos gráficos funcionais para entender o negócio agora.

A sintaxe é sempre a mesma:

1. **Onde está o X?** (Eixo horizontal)
2. **Onde está o Y?** (Eixo vertical)
3. **Qual o tipo?** (Barras, Linha, Box, Dispersão)

Gráfico de Barras: Quem ganha o jogo?

No Capítulo 4, vimos que temos categorias diferentes de produtos. Qual delas traz mais dinheiro para a empresa?

```
# Analisando Faturamento por Categoria
app.plotar(
    x="categoria",
    y="total_venda",
    tipo="barras",
    titulo="Receita Total por Categoria"
)
```

O Sanice é inteligente. Ele percebe que `categoria` é texto e `total_venda` é número. Automaticamente, ele agrupa os dados e soma os valores. Você não precisa fazer um `groupby()` antes. Ao rodar isso, uma janela interativa se abrirá. Você verá que, embora vendamos muitos "Cabos HDMI", é a categoria "Hardware" que paga as contas.

Gráfico de Linha: A Jornada do Tempo

Vendas são dinâmicas. Um número total não diz se a empresa está crescendo ou falindo. Para ver o tempo, usamos `tipo="linha"`.

```
# Analisando a tendência temporal
app.plotar(
    x="data_venda",
    y="total_venda",
    tipo="linha",
    titulo="Evolução Diária das Vendas"
)
```

Lembra que convertemos `data_venda` para objeto de data no capítulo anterior? O Sanice usa isso para ordenar o eixo X cronologicamente. Se houver buracos (dias sem vendas), a linha mostrará a queda.

Mapa de Calor: O Detector de Vazamentos

Este é o gráfico favorito dos Cientistas de Dados. O Mapa de Calor (Heatmap) mostra a Correlação entre todas as colunas numéricas.

- **Cor Quente (Vermelho/1.0):** As colunas sobem juntas. (Ex: `subtotal` e `total_venda`).
- **Cor Fria (Azul/-1.0):** Quando uma sobe, a outra desce.
- **Cor Neutra (Cinza/0.0):** Não tem nada a ver uma com a outra.

```
# A Matriz da Verdade
app.mapa_calor( )
```

Por que isso é vital?

Observe se existe algum quadrado vermelho brilhante (correlação perto de 1.0) entre colunas que não deveriam ser idênticas.

Por exemplo, você verá uma correlação fortíssima entre `comissao` e `total_venda`. Isso faz sentido matemático (a comissão é 5% do total), mas para a Inteligência Artificial, isso é um perigo chamado **Vazamento de Dados (Data Leakage)**.

Se deixarmos a coluna `comissao` entrar no treino, a IA vai "decorar" o resultado em vez de aprender a calcular. O Mapa de Calor serve justamente para isso: nos avisar quais colunas devemos remover antes do treino para evitar modelos viciados.

Prática: Olhando antes de saltar

`(cap4-5_visualizacao.py)`

Vamos criar o script **`chapters/cap4-5_visualizacao.py`**. O objetivo aqui não é salvar nada, é apenas abrir janelas na sua tela para que você, o humano, entenda o que está acontecendo antes de passar a bola para a máquina.

```
import os
from sanice import Sanice

# Caminhos
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
ENTRADA = os.path.join(BASE_DIR, "data", "processed", "vendas_com_features.csv")

print("--- Abrindo os Olhos para os Dados ---")
app = Sanice(ENTRADA)
```

```
# 1. Visão de Negócio (Barras)
# Pergunta: Qual categoria fatura mais?
app.plotar(
    x="categoria",
    y="total_venda",
    tipo="barras",
    titulo="Receita por Categoria"
)

# 2. Visão Temporal (Linha)
# Pergunta: Tivemos algum pico de vendas estranho?
app.plotar(
    x="data_venda",
    y="total_venda",
    tipo="linha",
    titulo="Tendência Janeiro 2025"
)

# 3. Visão Estatística (Boxplot)
# Pergunta: Existem vendas com valores absurdos (Outliers)?
# O Boxplot desenha uma "caixa" onde está a maioria dos
dados.
# Pontos fora da caixa são anomalias.
app.plotar(
    x="categoria",
    y="total_venda",
    tipo="box",
    titulo="Distribuição de Preços"
)

print("Gráficos gerados! (Verifique as janelas pop-up ou a
pasta 'plots')")
```

O que procurar?

Ao rodar este script, procure por:

1. **Categorias Dominantes:** Se uma categoria representa 90% das vendas, seu modelo será enviesado para ela.
2. **Picos Temporais:** Se houver um dia com vendas zero, pode ser um erro de sistema (bug) e não realidade.
3. **Outliers:** No Boxplot, se você vir um pontinho solitário lá em cima (uma venda de R$ 50.000 quando a média é R$ 200), considere usar o `.tratar_outliers()` que vimos no Capítulo 4.

Agora que você viu a realidade, podemos finalmente deixar a IA aprender com ela. Vamos para o torneio de modelos.

Capítulo 5

O Torneio de Modelos (AutoML 2.0)

"Não escolha o modelo favorito. Escolha o modelo que vence."

Até agora, preparamos o terreno. Lemos, limpamos e criamos novas colunas. Agora, queremos prever o futuro.

No jeito tradicional de fazer Data Science, você gastaria horas testando algoritmos:

1. *"Vou testar uma Regressão Linear..."* (Deu erro porque tem texto na coluna).
2. *"Agora vou testar uma Random Forest..."* (Demorou muito).
3. *"Agora vou tentar XGBoost..."* (Preciso instalar outra biblioteca).

O **Sanice** elimina essa indecisão com o conceito de **Torneio de Modelos**.

Quando você roda o comando `.auto_ml()`, você não está treinando uma IA. Você está treinando **três** e fazendo elas lutarem entre si para ver quem acerta mais.

Os Competidores da Arena

O Sanice coloca na arena três gladiadores clássicos do Scikit-Learn, cobrindo as principais abordagens matemáticas:

1. **O Veloz (Linear/Logistic Regression):**
    - Simples e extremamente rápido.
    - Ótimo para encontrar tendências óbvias (ex: quanto mais quartos, mais cara a casa).
    - **Quando vence:** Quando os dados têm uma relação linear clara e poucos ruídos.
2. **O Robusto (Random Forest):**
    - Cria centenas de "árvores de decisão" e tira a média.
    - Não liga para dados bagunçados ou não-lineares.
    - **Quando vence:** Na maioria dos casos do mundo real com dados complexos.
3. **O Especialista (Gradient Boosting):**
    - Aprende com os erros. Ele cria uma árvore, vê onde errou, e cria a próxima árvore focada apenas em corrigir aquele erro.
    - **Quando vence:** Em competições de precisão (como Kaggle), geralmente superando a Random Forest por uma margem pequena.

Você não precisa escolher. O Sanice treina os três, compara a métrica de sucesso (Acurácia ou R^2) e devolve apenas o campeão salvo no seu disco.

Classificação vs. Regressão

A única coisa que o Sanice exige de você é que diga **qual é a natureza do problema**.

1. Classificação (`tipo="classificacao"`)

Você quer prever uma **Categoria** (Texto ou Rótulo).

- O cliente vai comprar? (Sim/Não)
- Qual o tipo do produto? (Periférico/Hardware/Software)
- **Métrica de Vitória:** Acurácia (Quem acertou mais % de respostas).

2. Regressão (`tipo="regressao"`)

Você quer prever um **Número** (Valor Contínuo).

- *Qual será o total da venda?* (R$ 1.500,00)
- *Quantas unidades venderemos?* (350)
- **Métrica de Vitória: R^2** Score (Quão bem a linha do modelo se ajusta aos dados. 1.0 é perfeição).

Preparação Automática (O Trabalho Sujo Invisível)

Antes do torneio começar, o Sanice faz nos bastidores o que chamamos de "Pré-processamento Silencioso":

1. **Encoder (Get Dummies):** Algoritmos de IA só entendem números. O Sanice pega colunas de texto (como `estado`:

"SP", "RJ") e as transforma em colunas numéricas binárias (`estado_SP`: 1, `estado_RJ`: 0).

2. **Remoção de Datas Cruas:** Datas como "2025-01-01" confundem a matemática. O Sanice as remove automaticamente antes do treino, assumindo que você já extraiu o que queria delas no Capítulo 4 (Dia da Semana, Mês, etc.).
3. **Split Treino/Teste:** Ele separa automaticamente 20% dos dados para a "Prova Final", garantindo que o modelo não "decore" as respostas.

Tudo isso acontece em milissegundos antes da primeira árvore ser plantada.

Prática: Prevendo o Total da Venda

`(cap05_automl.py)`

Vamos testar isso no nosso dataset **`vendas_com_features.csv`**. Queremos prever o `total_venda` baseado nas outras colunas (Produto, Estado, Frete, etc.).

Como `total_venda` é um número, usaremos **Regressão**.

Crie o arquivo `chapters/cap05_automl.py`:

```
import os
from sanice import Sanice

# Caminhos
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))

ENTRADA = os.path.join(BASE_DIR, "data", "processed", "vendas_com_features.csv")
```

```
MODELO_SAIDA = os.path.join(BASE_DIR, "data",
"processed", "meu_primeiro_modelo.pkl")

print("--- Iniciando o Grande Torneio de IA ---")

# 1. Carregar os dados já enriquecidos
app = Sanice(ENTRADA)

(app
# [ANTI-CHEAT] Seleção Estratégica de Features
# Removemos os 'id' que não usaremos na previsão.
# Motivo: A comissão é calculada baseada no total. Se a IA ver
isso,
# ela "decora" a resposta (Vazamento) e não aprende a calcular
de verdade.
   .selecionar_colunas([
      "subtotal",
      "frete_formato_sujo",
      "total_venda" # Alvo
   ])
 # O Sanice vai treinar 3 modelos (Linear, Random Forest,
Gradient Boosting),
# comparar quem tem o melhor R² Score e salvar apenas o
vencedor.
   .auto_ml(
      alvo="total_venda",        # O que queremos prever
      tipo="regressao",          # Número ou Categoria
      salvar_modelo=MODELO_SAIDA  # Onde salvar o
cérebro campeão
   )
)
```

O que esperar no Terminal?

Ao rodar, você verá algo assim:

```
[AUTO-ML] Iniciando treinamento para prever: 'total_venda'...
[AUTO-ML] Avaliando 3 modelos (Linear, RF, Gradient)...
[RESULTADO] Melhor modelo: LinearRegression | R² Score: 1.0000
   Modelo salvo em: .../meu_primeiro_modelo.pkl
```

Por que `LinearRegression` venceu com Score 1.0 (Perfeito)? Porque no Capítulo 4 nós criamos a coluna `total_venda` usando uma fórmula matemática exata (`valor * quantidade + frete`). A IA "descobriu" essa fórmula matemática simples. É a prova de que a máquina aprendeu a regra do jogo!

> ***Nota:*** *Se estivéssemos prevendo algo caótico, como "Preço de Ações", o Score seria menor e provavelmente o* `GradientBoosting` *venceria.*

Agora você tem um arquivo `.pkl` no seu computador. Esse arquivo é um **cérebro congelado**. No próximo capítulo, vamos aprender a "descongelar" esse cérebro para fazer previsões em novos clientes.

Capítulo 6

Previsão na Prática

"Treinar é fácil. Difícil é colocar em campo."

Você tem um arquivo *meu_primeiro_modelo.pkl* na sua pasta. O que é isso? É um arquivo binário (serializado) que contém toda a matemática, as árvores de decisão e os pesos que o seu computador aprendeu durante o torneio do Capítulo 5.

É como se tivéssemos congelado o cérebro do nosso Cientista de Dados em um pen-drive.

Agora, imagine que chegou um novo pedido no site. O cliente escolheu um produto de **R$ 800,00**, vai levar **3 unidades** e o frete deu **R$ 20,00**. Você quer saber se o cálculo do sistema bate com a previsão da IA (ou, num cenário mais complexo, prever o Churn desse cliente).

Para isso, usamos dois comandos: `.carregar_ia()` e `.prever()`.

Carregando Cérebros Digitais (`.carregar_ia`)

Diferente dos capítulos anteriores, onde carregávamos um CSV de treino, agora vamos carregar **novos dados** (o cliente novo) e aplicar a **velha inteligência** (o modelo salvo).

```
from sanice import Sanice, pd

# 1. Dados do Novo Cliente (Simulação)
# Note que não temos a coluna 'total_venda', pois é ela que queremos prever!
novo_cliente = pd.DataFrame([{
    "valor": 800.00,
    "quantidade": 3,
    "frete_formato_sujo": 20.0,
    "estado": "SP"  # A IA aprendeu que Estado influencia? Veremos.
}])

# 2. Iniciamos o Sanice com esses dados novos
app = Sanice(novo_cliente)

# 3. Carregamos o Cérebro
app.carregar_ia("data/processed/meu_primeiro_modelo.pkl")
```

Quando você roda `.carregar_ia()`, o Sanice lê o arquivo e configura o ambiente exatamente como estava no momento do treino. Ele lembra quais colunas existiam, qual era a média para normalização e qual algoritmo venceu o torneio.

Fazendo a Previsão (`.prever`)

Agora a mágica. O método `.prever()` pega os dados do `app`, prepara-os (fazendo o mesmo *get_dummies* e tratamento de colunas que fizemos no treino) e passa pelo modelo.

```
# Gera a previsão e salva numa nova coluna chamada 'previsao_ia'
app.prever("previsao_ia")
```

```
# Vamos ver o resultado?
app.ver()
```

Se o modelo aprendeu corretamente a fórmula matemática (`Valor * Qtd + Frete`), ele deve responder algo muito próximo de: 800×3+20=2420

Se a previsão for `2420.00`, parabéns! Você criou uma IA que aprendeu aritmética básica sozinha. Se fosse um problema de **Classificação** (ex: Fraude), a saída seria **`0`** (Não Fraude) ou **`1`** (Fraude).

Métricas de Sucesso: Posso confiar?

No log do Capítulo 5, você viu termos como "Acurácia" ou "R² Score". O que significam?

- **R² Score (Regressão):** Varia de 0 a 1.
  - **1.0:** Perfeição. O modelo previu o valor exato.
  - **0.9:** Excelente.
  - **0.5:** O modelo acerta a tendência, mas erra os valores exatos.
  - **< 0:** O modelo é pior do que "chutar a média". Jogue fora.
- **Acurácia (Classificação):** % de acertos.
  - **0.95:** Acertou 95 de 100 casos. Ótimo.
  - **0.50:** Cara ou Coroa. Se for um problema binário (Sim/Não), seu modelo é inútil.

Nota de Rodapé: *Cuidado com acurácias de 100% (1.0) em dados complexos. Geralmente indica Overfitting (o modelo decorou a prova em vez de aprender a matéria). No nosso exemplo de vendas, 1.0 é aceitável porque a relação é puramente matemática e exata.*

Vamos Brincar?

(`cap06_previsao.py`)

Crie o arquivo `chapters/cap06_previsao.py`. Vamos criar 3 clientes fictícios diferentes para ver como a IA reage.

```
import os
from sanice import Sanice, pd

BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
MODELO_PATH = os.path.join(BASE_DIR, "data", "processed", "meu_primeiro_modelo.pkl")

print("--- Testando a IA com Engenharia de Features ---")

# Criamos os dados JÁ com o subtotal calculado (simulando o front-end enviando)
dados_novos = pd.DataFrame([
    {"subtotal": 50.0 * 1, "frete_formato_sujo": 10.0, "categoria": "Periféricos"}, # Mouse
    {"subtotal": 1200.0 * 2, "frete_formato_sujo": 50.0, "categoria": "Monitores"}, # Monitor
    {"subtotal": 4500.0 * 3, "frete_formato_sujo": 100.0, "categoria": "Hardware"}  # Placa
])

# Adicionamos colunas auxiliares apenas para nossa validação visual (a IA não usará)
dados_novos["valor"] = [50.0, 1200.0, 4500.0]
dados_novos["quantidade"] = [1, 2, 3]

app = Sanice(dados_novos)

(app
    .carregar_ia(MODELO_PATH)

    .prever("previsao_valor_total")
    .ver(titulo="Vitória da Engenharia de Dados")
```

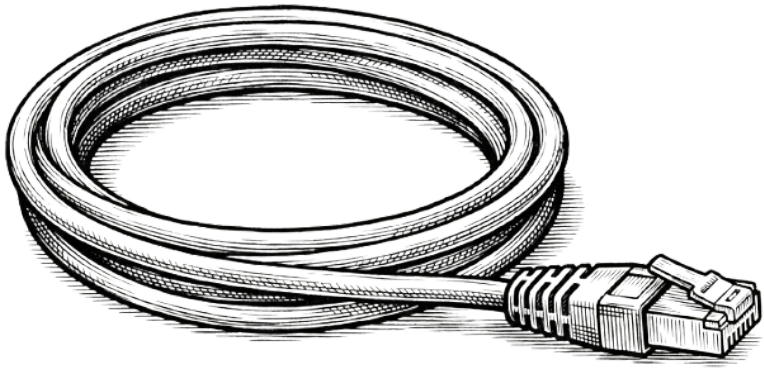
```
)

print("\n--- Validação Final ---")
for i, linha in app.pegar_dataframe().iterrows():
    real = linha['subtotal'] + linha['frete_formato_sujo']
    previsto = linha['previsao_valor_total']
    diff = abs(real - previsto)
    print(f"Caso {i+1}: Real R$ {real:.2f} | IA R$ {previsto:.2f} |
Erro: {diff:.4f}")
```

O que você deve fazer agora:

1. Rode esse script.
2. Veja se a **Diferença** é 0.0000 (ou muito próxima de zero, tipo `0.000000001` devido a pontos flutuantes).
3. Se for zero, sua IA está calibrada e pronta para produção!

Isso encerra a **Parte 3** do livro. Você treinou e usou uma IA. Agora, na **Parte 4**, vamos sair do "brinquedo" e ir para o mundo real: Banco de Dados SQL, MongoDB e criar uma API (Site) para essa IA.

PARTE 4



DO NOTEBOOK PARA O MUNDO

Capítulo 7

O Poliglota de Dados (SQL & NoSQL)

"CSV é vida, mas Banco de Dados é produção."

Até agora, trabalhamos com arquivos `.csv` e `.pkl`. Funciona muito bem para estudos. Mas no mundo corporativo, os dados não chegam em um arquivo bonitinho por e-mail. Eles vivem dentro de fortalezas, os **Bancos de Dados**.

- O financeiro usa **PostgreSQL** ou **Oracle**.
- O site novo usa **MongoDB** ou **Firebase**.
- O legado da empresa roda em **SQL Server** ou **MySQL**.

O Sanice foi desenhado para ser "Agnóstico de Banco". Isso significa que você não precisa decorar 5 bibliotecas diferentes (`psycopg2`, `pymysql`, `pymongo`). Você só precisa de uma **URL de Conexão**.

Lendo direto da Fonte (*`Sanice.de_sql`*)

Em vez de pedir para a TI extrair um CSV para você, o Sanice permite conectar direto na fonte.

Usamos o método de classe `Sanice.de_sql(url, query)`. Ele usa a biblioteca **SQLAlchemy** por baixo dos panos, garantindo conexões robustas.

As URLs Mágicas (Connection Strings)

O segredo está no primeiro parâmetro. É uma string que diz quem é o banco, onde ele está e a senha.

- **PostgreSQL:** `postgresql://usuario:senha@localhost:5432/meubanco`
- **MySQL (com nosso ajuste pymysql):** `mysql+pymysql://usuario:senha@localhost/meubanco`
- **SQLite (Arquivo local):** `sqlite:///meu_arquivo.db`

O Código Exemplo

```
from sanice import Sanice
# Conectando no banco para pegar vendas
# Note: Isso retorna uma instância Sanice pronta para limpeza!
app = Sanice.de_sql(
    url_conexao="sqlite:///loja.db",
    query="SELECT * FROM vendas WHERE data >= '2025-01-01'"
)
# A partir daqui, é vida normal. O Sanice não liga se veio do SQL ou CSV.
app.ver(5)
```

Exportando para o Futuro (`.exportar_mongo`)

Imagine que você gerou as previsões de IA. Agora, o aplicativo móvel quer exibir isso para o usuário. Eles não usam SQL. Eles usam **MongoDB** (NoSQL), porque é rápido e flexível (JSON).

O método `.exportar_mongo()` converte seu DataFrame em dicionários JSON e empurra para a coleção especificada. Se a coleção não existir, o Mongo cria na hora.

```
app.exportar_mongo(
    uri="mongodb://localhost:27017",
    database="app_vendas",
    collection="previsoes_jan_2025"
)
```

Prática: O Pipeline de Integração

(`cap07_integracao.py`)

Como provavelmente você não tem um servidor PostgreSQL rodando na sua máquina agora, vamos usar o **SQLite** (que cria um banco num arquivo simples `.db`) para simular o "Sistema Legado".

Crie o arquivo `chapters/cap07_integracao.py`.

Neste script, faremos uma simulação completa de Engenharia de Dados:

1. **Setup:** Pegaremos nosso CSV pronto (do Cap 4) e salvaremos num banco SQL (fingindo ser o sistema da empresa).

2. **Extração:** Leremos desse banco SQL usando `Sanice.de_sql`.
3. **Carga (Load):** Tentaremos exportar para o MongoDB.

```
import os
from sanice import Sanice

# Caminhos
BASE_DIR =
os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
CSV_ENTRADA = os.path.join(BASE_DIR, "data",
"processed", "vendas_com_features.csv")
BANCO_SQLITE = os.path.join(BASE_DIR, "data",
"processed", "sistema_loja.db")

# Connection String para SQLite (simulando um banco real)
URL_SQL = f"sqlite:///{BANCO_SQLITE}"

print("--- Iniciando Simulação de Engenharia de Dados (SQL
& NoSQL) ---")

# 1. PREPARAÇÃO DO AMBIENTE (Simulando o Sistema
Legado)
# Vamos pegar nosso CSV perfeito e forçar ele para dentro de
um banco SQL
# Só para termos de onde ler depois.
print(f"[SETUP] Criando banco SQL simulado em:
{BANCO_SQLITE}...")

app_csv = Sanice(CSV_ENTRADA) # Exportar SQL: Cria a
tabela e insere os dados
app_csv.exportar_sql(URL_SQL, "tabela_vendas_2025",
modo="replace")

# 2. O CENÁRIO REAL (Leitura do SQL)
# Imagine que é um novo dia. Você não tem CSV.
# Você conecta no banco para ler apenas vendas altas.
```

```
print("\n[ETAPA 1] Conectando no Banco de Dados via Sanice...")

app_sql = Sanice.de_sql(
    url_conexao=URL_SQL,
    query="SELECT * FROM tabela_vendas_2025 WHERE total_venda > 1000"
)

if app_sql:
    # Mostra que os dados vieram do SQL intactos
    (app_sql
        .ver(3, titulo="Dados extraídos do SQL (Filtrados)")
        .resumo_estatistico()
    )

    # 3. EXPORTAÇÃO (Tentativa NoSQL - MongoDB)
    # Se você tiver MongoDB instalado, isso funcionará e criará o banco.
    # Se não, o Sanice capturará o erro e avisará, sem travar o script.
    print("\n[ETAPA 2] Tentando exportar para Data Lake (MongoDB)...")
    (app_sql
        .exportar_mongo(
            uri="mongodb://localhost:27017",
            database="datalake_sanice",
            collection="vendas_high_ticket"
        )
    )
else:
    print("[ERRO] Não foi possível conectar ao SQL.")
print("\n--- Fim do Capítulo 7 ---")
```

Capítulo 8

Sua IA no Ar: Criando uma API

"Se não tem API, não existe para o mundo."

Parabéns! Você tem um modelo de Inteligência Artificial treinado, validado e salvo no seu disco (`meu_primeiro_modelo.pkl`). Mas, neste momento, ele é apenas um arquivo "morto" no seu computador.

Se o time de vendas quiser usar sua previsão, eles têm que te ligar? Se o aplicativo da empresa quiser consultar o modelo, ele consegue acessar seu notebook? Não.

Para tornar sua IA útil para a empresa, precisamos fazer o **Deploy** (implantação). Vamos transformar esse arquivo em um **Microsserviço Web (API)**.

Uma **API** é como um garçom digital:

1. O cliente (Site/App) faz um pedido (envia os dados da venda).
2. A API leva o pedido para a cozinha (seu Modelo de IA).
3. A API devolve o prato pronto (a previsão de valor).

O Servidor de Uma Linha (`.servir_api`)

Em frameworks tradicionais (como Flask ou Django), você precisaria criar rotas, configurar servidores, serializar JSON e tratar erros. Seriam umas 30 a 50 linhas de código.

No Sanice, como já padronizamos a entrada e a saída nos capítulos anteriores, levantar a API custa apenas uma linha de código.

O método `.servir_api()`:

1. Levanta um servidor **FastAPI** (o mais rápido do mercado Python).
2. Cria automaticamente uma rota `/predict`.
3. Gera uma documentação interativa (Swagger UI).

O Código Final

(cap08_api.py)

Crie este arquivo. Ele é o ápice da nossa jornada: transformar ciência em produto.

```
import os
from sanice import Sanice

# Caminho do Modelo Campeão (Treinado no Cap 5)
BASE_DIR =
os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
MODELO_PATH = os.path.join(BASE_DIR, "data",
"processed", "meu_primeiro_modelo.pkl")

print("--- Iniciando Servidor de Previsão de Vendas (Sanice
API) ---")
print(f"Carregando cérebro digital de: {MODELO_PATH}")
```

```
# 1. Instanciar o Sanice em "Modo Servidor"
# Passamos None pois não vamos processar CSVs agora, apenas servir o modelo.
app = Sanice(None)

# 2. Carregar a Inteligência
app.carregar_ia(MODELO_PATH)

# 3. O Comando Mágico
# Isso vai iniciar o servidor e ficar aguardando conexões.
app.servir_api()
```

Testando sua API no Navegador

Ao rodar o script acima, o terminal ficará "preso" (isso é normal, o servidor está rodando) e mostrará uma mensagem como: `[API] Servidor iniciado em http://127.0.0.1:8000/docs`

Clique nesse link ou abra no seu navegador. Você verá uma tela azul profissional. Essa é a **Swagger UI**, uma interface de teste automática que o Sanice gerou para você.

Fazendo uma Previsão Real

1. Clique na barra verde **POST /predict**.
2. Clique no botão **Try it out** (canto direito).
3. No campo de texto (Request Body), você deve enviar o JSON com os dados.

Atenção: Lembre-se da lição do Capítulo 6! Nossa IA aprendeu a usar o `subtotal`, não o valor unitário. Envie os dados corretos:

```
{

  "subtotal": 50.0,
  "frete_formato_sujo": 10.0
}
```


Clique no botão azul **Execute**.
Role para baixo até **Server response**. Você verá:


```
{
  "predicao": 60.0
}
```


Qualquer sistema do mundo que enviar esse JSON para o seu IP receberá essa resposta. Você acabou de colocar uma Inteligência Artificial em produção.

PARTE 5

PROJETO DE CONCLUSÃO

Capítulo 9

Mordor Inc. vs Airbnb

"Na teoria, a teoria e a prática são iguais. Na prática, não."

A **Mordor Inc.** está expandindo seus domínios para o Rio de Janeiro. O CEO, Sauron, tem um problema logístico grave: ele precisa alocar milhares de Orcs, Uruk-hais e os 9 Nazgûl, mas o orçamento de moedas de ouro está curto.

Ele contratou você para usar o **Sanice** com dois objetivos:

1. **Analítica (O Olho):** Entender o terreno. Onde é barato? Onde cabe mais gente?
2. **Preditiva (O Palantír):** Criar uma IA para precificar automaticamente novos esconderijos.

Parte 1: Análise Exploratória

(cap09_pt1.py)

Antes de invadir, precisamos de inteligência. O Conselho de Administração de Mordor fez 3 perguntas estratégicas:

1. *Os Nazgûl exigem privacidade. Qual a diferença de preço entre alugar uma casa inteira ou apenas um quarto?*
2. *Precisamos economizar. Quais são os 10 bairros mais baratos para amontoar os exércitos?*

3. *Orcs dormem empilhados. Existe relação entre o número de camas e o preço?*

Desafio a pessoa Leitora: Antes de revelarmos os segredos ocultos nos dados, chame o **Sanice** e tente responder a essas perguntas por conta própria. Que ferramentas do framework você usaria para visualizar essas distribuições? Tente escrever o script de análise antes de olhar a solução abaixo.

…

Conseguiu? Agora, vejamos como o **Sanice** responde a essas perguntas estratégicas em poucas linhas de código.

```
import os
from sanice import Sanice, pd

# --- caminhos ---
BASE_DIR =
os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
ARQUIVO_AIRBNB = os.path.join(BASE_DIR, "data", "raw",
"listings.csv")

print("--- Mordor Inc: Setor de Inteligência Logística ---")

#  ingestão e limpeza
print("[1/2] Limpando dados brutos...")
app = Sanice(ARQUIVO_AIRBNB)


(app
   .selecionar_colunas([
```

```
        "price", "room_type", "neighbourhood_cleansed",
        "accommodates", "beds"
    ])
    .criar_coluna("price", lambda row:
float(str(row["price"]).replace("$", "").replace(",", "")))
    .remover_nulos("preencher", 0)
    .filtrar("price > 0 and price < 5000")
)

print("Dados higienizados. Gerando relatórios via Sanice...")
# ---------------------------------------------------------------
# PERGUNTA 1: Relação Camas x Preço (Scatter)
# ---------------------------------------------------------------
(app
    .plotar(
        tipo="scatter",
        x="beds",
        y="price",
        hue="room_type",
        titulo="Camas vs Preço (Por Tipo de Acomodação)"
    )
)
# ---------------------------------------------------------------
# PERGUNTA 2: Custo por Privacidade (Barras)
# ---------------------------------------------------------------
print("Gerando comparativo de quartos...")

(Sanice(app.df)
    .agrupar(
        ["room_type"],
        "price",
        "media"
    )
    .ordenar("price", ascendente=True)
    .plotar(
        tipo="bar",

        x="room_type",
        y="price",
```

```
        titulo="Preço Médio por Tipo de Quarto"
    )

)
# ----------------------------------------------------------------
# PERGUNTA 3: Top Bairros Mais Baratos (Barra Horizontal)
# ----------------------------------------------------------------
print("Mapeando territórios baratos...")

(Sanice(app.df)
    .agrupar(
        ["neighbourhood_cleansed"],
        "price",
        "media"
    )
    .ordenar("price", ascendente=True)
    .plotar(
        tipo="bar",
        x="neighbourhood_cleansed",
        y="price",
        titulo="Média de Preço por Bairro (Menor para Maior)"
    )
)
```

Parte 2: A Previsão

(cap09_pt2.py)

Agora que entendemos o terreno, Sauron quer saber o futuro. A empresa encontrou dois novos imóveis potenciais. Eles não têm preço listado. O seu modelo de IA (O Palantír) deve dizer quanto é justo pagar por eles.

- **Cenário A (Toca dos Uruk-hai):** Um apartamento grande em Copacabana para 6 soldados.

- **Cenário B (Torre de Saruman):** Um quarto privado na Barra para um Mago.

Desafio a pessoa Leitora: Tente definir as variáveis e chamar a função de previsão antes de consultar a solução abaixo.

…

Conseguiu? No próximo código, analisamos como o **Sanice** trata esse cenário.

```
import os
from sanice import Sanice, pd

# --- caminhos ---
BASE_DIR =
os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
ARQUIVO_AIRBNB = os.path.join(BASE_DIR, "data", "raw",
"listings.csv")
MODELO_FINAL = os.path.join(BASE_DIR, "data",
"processed", "palantir_v1.pkl")

print("\n--- Parte 2: Ligando o Palantír (AutoML) ---")

# 1. Preparação  (ETL para a IA)
# Precisamos limpar os dados novamente para entregar "comida
limpa" para o cérebro da IA.
app = Sanice(ARQUIVO_AIRBNB)

(app
   .selecionar_colunas([
      "price", "room_type", "neighbourhood_cleansed",


      "accommodates", "beds", "bedrooms",
"number_of_reviews"
   ])
```

```
    .criar_coluna("price", lambda row:
float(str(row["price"]).replace("$", "").replace(",", "")))
    .remover_nulos("preencher", 0)
    # Importante: A IA aprende melhor sem dados errados
(preços zerados ou mansões absurdas)
    .filtrar("price > 0 and price < 5000")
)

# 2. Comparação de Modelos (Treinamento)
print("[Treinamento] Invocando espíritos da máquina (Isso
pode levar alguns segundos)...")

# Aqui a mágica acontece. O Sanice vai:
# 1. Transformar texto (Copacabana) em números.
# 2. Testar 3 algoritmos diferentes.
# 3. Escolher o melhor e salvar no arquivo .pkl
(app
    .auto_ml(
        alvo="price",          # O que queremos descobrir?
        tipo="regressao",      # Regressão = Prever um número
(Preço)
        salvar_modelo=MODELO_FINAL
    )
)

print(f"Palantír calibrado e salvo em: {MODELO_FINAL}")

# 3. A profecia (Simulação)
print("\n--- Consultoria para o Senhor do Escuro ---")

# Criamos os dados dos novos imóveis que Sauron quer alugar
novas_bases = pd.DataFrame([
    {
        "codigo": "BASE_URUK_HAI",
        "neighbourhood_cleansed": "Copacabana",
        "room_type": "Entire home/apt",

        "accommodates": 6,
        "beds": 6,
        "bedrooms": 2,
```

```
        "number_of_reviews": 15 # Lugar movimentado
    },
    {
        "codigo": "TORRE_SARUMAN",
        "neighbourhood_cleansed": "Barra da Tijuca",
        "room_type": "Private room",
        "accommodates": 1, # Apenas o Mago
        "beds": 1,
        "bedrooms": 1,
        "number_of_reviews": 0 # Lugar discreto
    }
])

# Invocamos o Oráculo apenas para ler o futuro
oraculo = Sanice(novas_bases)

# Carregamos o cérebro que acabamos de treinar
oraculo.carregar_ia(MODELO_FINAL)

# Geramos a previsão
oraculo.prever("preco_justo_ouro")

# Apresentamos o resultado
print("\n--- RELATÓRIO FINAL ---")
print(oraculo.df[["codigo", "neighbourhood_cleansed",
"preco_justo_ouro"]])
```

Conclusão: A Decisão do Lorde das Trevas

Ao rodar o script final, o terminal não exibiu apenas números; ele exibiu **estratégia**.

O *Sanice* permitiu que, em menos de 50 linhas de código, passássemos por todo o ciclo de vida de um projeto de Ciência de Dados profissional:

1. **ETL:** Lemos dados brutos, limpamos a sujeira (símbolos de moeda, nulos) e removemos o que não fazia sentido (imóveis de preço zero).
2. **Analytics:** Respondemos perguntas de negócio com gráficos claros. Sauron descobriu que alugar quartos privados na Zona Oeste é muito mais barato do que mansões na Zona Sul.
3. **Machine Learning:** Criamos um "Palantír" (Modelo Preditivo) que aprendeu com o passado para precificar o futuro, sem precisarmos gastar horas configurando hiperparâmetros.

No mundo real, seu chefe (provavelmente menos assustador que Sauron, mas igualmente exigente) não quer saber se você usou *XGBoost* ou *RandomForest*. Ele quer saber: **"Quanto custa?"** e **"Onde investimos?"**.

A biblioteca *Sanice* removeu a barreira técnica para que você pudesse focar na **inteligência do negócio**. A invasão de Mordor ao Rio de Janeiro agora é baseada em dados, não em palpites. E é exatamente isso que separa um programador de códigos de um Cientista de Dados Ágil.

"Não é sobre o quão complexo é o seu martelo, mas sobre o quão bem você constrói a casa."

Você agora tem o martelo. Vá construir.

Capítulo 10

O Oráculo de Bolso (Deploy)

"Tudo o que temos de decidir é o que fazer com o tempo que nos é dado.**"** — *Gandalf (O Senhor dos Anéis: A Sociedade do Anel)*

Imagine que passou uma semana. Os dados de treinamento (`listings.csv`) foram arquivados. O computador foi desligado. Mas a inteligência que criamos continua viva.

Chamamos isso de **Deploy** ou "Colocar em Produção".

Você não precisa re-treinar a IA toda vez que quiser uma resposta. Você apenas carrega o arquivo `.pkl` (o Palantír) que salvamos. Ele contém todas as regras matemáticas congeladas.

Hoje, um batedor orc chegou correndo com informações de três possíveis esconderijos novos na Zona Norte. Ele precisa saber o preço justo **agora**. Não temos tempo para abrir gráficos.

Desafio a pessoa Leitora: Como você carregaria apenas o "cérebro" da nossa IA para interrogar esses novos dados? Tente estruturar o script de consulta e invocar o oráculo antes de verificar a solução abaixo.

…

Solução:

(consultoria_rapida.py)

Este código representa o "lado B" do Data Science: quando o modelo sai do laboratório e vai para o campo de batalha. Note que aqui não usamos o arquivo CSV original, apenas o `.pkl`.

```
import os
from sanice import Sanice, pd

# --- caminhos ---
BASE_DIR =
os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
# Apontamos apenas para o arquivo .pkl que geramos no passo
anterior
CAMINHO_DO_PALANTIR = os.path.join(BASE_DIR, "data",
"processed", "palantir_v1.pkl")

print("--- Consultoria Rápida de Mordor ---")

if not os.path.exists(CAMINHO_DO_PALANTIR):
    print("ERRO: O Palantír não foi encontrado! Rode o script
de treinamento primeiro.")
    exit()

# 1. Entrada de Dados
# O orc trouxe esses dados anotados em um pedaço de pão
lembas.
# Note: As colunas devem ser as mesmas usadas no
treinamento!
novos_esconderijos = pd.DataFrame([
    {
        "nome": "Bunker Meier",
        "neighbourhood_cleansed": "Méier",
        "room_type": "Entire home/apt",
        "accommodates": 4,
        "beds": 3,
```

```
        "bedrooms": 2,
        "number_of_reviews": 5
    },
    {
        "nome": "Toca Pavuna",
        "neighbourhood_cleansed": "Pavuna",
        "room_type": "Shared room", # Quarto compartilhado é
mais barato
        "accommodates": 10,
        "beds": 10,
        "bedrooms": 1,
        "number_of_reviews": 0
    },
    {
        "nome": "Cobertura Leblon (Espião)",
        "neighbourhood_cleansed": "Leblon",
        "room_type": "Entire home/apt",
        "accommodates": 2,
        "beds": 1,
        "bedrooms": 1,
        "number_of_reviews": 50
    }
])

print(f"[INPUT] Recebidos {len(novos_esconderijos)} novos
locais para avaliação.")

# Iniciamos o Sanice com os dados novos
app = Sanice(novos_esconderijos)

# Carregamos a inteligência pré-treinada
# Isso é instantâneo, pois não há cálculos pesados, apenas
leitura de memória.
app.carregar_ia(CAMINHO_DO_PALANTIR)

# 3. a previsão
print("[PROCESSANDO] Consultando os astros...")

app.prever("sugestao_preco")
```

```
# 4. RESULTADO PARA TOMADA DE DECISÃO
# Formatamos a saída para ficar bonita no terminal
print("\n" + "="*40)
print(" TABELA DE PREÇOS RECOMENDADOS (R$)")
print("="*40)

# Selecionamos apenas o que interessa para exibir
resultado = app.df[["nome", "neighbourhood_cleansed",
"sugestao_preco"]]

# Iteramos para imprimir linha por linha (opcional, só por
estética)
for index, linha in resultado.iterrows():
    print(f"{linha['nome']}
({linha['neighbourhood_cleansed']})")
    print(f"Preço Previsto: R$ {linha['sugestao_preco']:.2f}")
    print("-" * 40)
```

Você pode estar se perguntando: *"Por que eu preciso salvar esse arquivo? Não posso rodar o código de novo?"*

Poder, pode. Mas imagine se, toda vez que um orc perguntasse o preço, você tivesse que ler 40.000 linhas de Excel e refazer todos os cálculos matemáticos. O exército travaria.

Ao salvar o modelo (o arquivo `.pkl`), ganhamos três superpoderes:

1. **Portabilidade (O Cérebro de Bolso):** Você pode enviar apenas o arquivo `.pkl` (que tem alguns Kilobytes) e o script de consulta por e-mail para outro computador. Ele vai funcionar **sem precisar do CSV original** de 300MB. O modelo já "aprendeu" o padrão; ele não precisa mais dos dados brutos.

2. **Velocidade (Reflexo vs. Estudo):** Enquanto o treinamento (`auto_ml`) demorou segundos ou minutos estudando o passado, a previsão (`prever`) demora **milissegundos**. É instantâneo. É assim que o Google te dá respostas em tempo real sem ter que ler a internet inteira a cada busca.
3. **Encapsulamento (A Regra Invisível):** O modelo salvo "lembra" sozinho que o Leblon vale mais que o Méier. Você não precisa escrever milhares de `if/else` no seu código (ex: `if bairro == 'Leblon': preco + 100`). O arquivo **`.pkl`** já contém essa inteligência matemática gravada nele.
4.

A Vantagem do Cérebro Persistente:

Com o **Sanice** e a persistência do Palantír, a Mordor Inc. não está apenas automatizando processos; ela está dobrando o tempo a seu favor.

Matematicamente, a eficiência do seu exército agora segue a lógica da economia de escala. Se T_{treino} é o tempo de estudo e $T_{inferencia}$ é o tempo de resposta, temos:

$$Gain = \left(1 - \frac{T_{inferencia}}{T_{treino}}\right) \times 100$$

Considerando que o treinamento de uma base de 40.000 linhas pode levar 60 segundos, e a previsão do Oráculo leva apenas 0.006 segundos, sua vantagem competitiva é de **99,99% em agilidade de resposta**.

Conclusão

Do Caos à Inteligência

"A perfeição não é alcançada quando não há mais nada a acrescentar, mas sim quando não há mais nada a retirar." — Antoine de Saint-Exupéry

Concluir este guia de campo é apenas o marco inicial da sua autonomia. O que começou com um arquivo de dados bruto, inconsistente e desorganizado, transformou-se em um sistema de inteligência capaz de gerar previsões precisas e rápidas. Ao longo deste livro, você não apenas aprendeu a operar o **Sanice**, mas também a adotar uma mentalidade de eficiência na Ciência de Dados.

A Vantagem da Automação

A verdadeira vantagem de utilizar um framework de **Interface Fluente** não é apenas escrever menos código, mas sim aumentar a sua taxa de iteração. Em Ciência de Dados, o sucesso é alcançado através de ciclos rápidos de experimentação. Ao utilizar o **Sanice** para remover o peso da sintaxe repetitiva, você foca sua energia na resolução de problemas de negócio.

O Próximo Passo é Seu

O **Sanice** não substitui o seu conhecimento — ele o potencializa. A ferramenta remove a barreira técnica para que você possa focar na inteligência que realmente gera valor. O código que você escreveu e as técnicas que você aprendeu aqui são agora o seu novo padrão de produtividade.

Convite à Contribuição

Obrigado por percorrer esta jornada e chegar até aqui. Este livro foi escrito para demonstrar que a Ciência de Dados não precisa ser um processo burocrático ou excessivamente complexo. O **Sanice** (Sistema Automatizado de Normalização, Inteligência Computacional e Estatística) nasceu de uma necessidade real de agilidade, focando em reduzir a barreira da sintaxe para priorizar a inteligência do negócio.

Contudo, o código que você viu nestas páginas é apenas o alicerce. O **Sanice** é um projeto **Open Source** (Código Aberto) e sua evolução contínua depende da colaboração de desenvolvedores e cientistas de dados que enfrentam os desafios do mundo real todos os dias.

Como você pode fazer parte?

Se você encontrou um bug, desenvolveu uma nova regra de limpeza ou simplesmente pensou em uma forma de tornar a documentação mais clara, o seu contributo é muito bem-vindo. Existem diversas formas de ajudar o projeto a crescer:

- **Reportar Problemas (Issues)**: Encontrou um comportamento inesperado ou um erro de conversão em algum dataset específico? Abra uma *issue* no repositório detalhando o caso.
- **Melhorar a Documentação**: Se você criou um exemplo prático ou um caso de uso interessante, compartilhe-o. Documentar é tão importante quanto codificar.
- **Desenvolver Novas Funcionalidades**: O Sanice foi projetado para ser extensível. Sinta-se à vontade para

implementar novos filtros, visualizações ou suporte a novos tipos de dados.

- **Divulgação e Feedback**: Utilize o framework em seus projetos pessoais ou profissionais e compartilhe sua experiência no LinkedIn ou em blogs técnicos.

Onde Encontrar o Projeto

O coração do Sanice bate no GitHub. Lá você encontrará o código-fonte completo, o histórico de versões e poderá interagir diretamente com o desenvolvimento da ferramenta.

O objetivo final é garantir que o Data Science continue sendo divertido e, acima de tudo, útil. Ao contribuir, você não apenas melhora uma ferramenta, mas ajuda a construir um ecossistema que respeita o tempo de quem desenvolve.

Acesse o repositório oficial:
https://github.com/wSanice/sanice

w**Sanice.**